Università
della
Svizzera
italiana

Faculty
of
Informatics

Bachelor Thesis

June 19, 2019

Android Malware Activation Strategies Comparison for Dynamic Detection

Alberto Vitoria Pascual

Abstract

Dynamic malware detection is performed by monitoring system parameters at runtime (i.e., behavior of applications is monitored as they run on the system). To collect data necessary for the development of such detection methods, applications need to be run in a controlled environment and malware need to be properly triggered. Some methods are totally random (i.e., the exerciser creates a predefined number of events), while some others are based on GUI models (i.e., generated events are generated by using a library of different user interfaces).

Goal of this project is to compare different methods for exercising applications, with the purpose of verifying that there is a significant difference between the methods. The project was performed by using already available malware samples and the comparison was performed by considering results obtained at USI.

Results were obtained sufficient to say that there is a difference in some of the features extracted while in others is less significant.

Keywords:android;malware;droidbot;dynamic detection;linux

Advisor:
Alberto Ferrante

Approved by the advisor on Date: 19-06-2019

Contents

1 Introduction

Android devices cover more than 50% of the market share making the Google OS the top target for malware developers. A recent Techcrunch article [5] revealed that over 200 apps with more than 150 million downloads were malware. Google's official figures indicated that more than 700,000 apps were removed from the play store last year. As this issue has increasingly frequented the news, it is sufficient to say that malware detection is an important concern in the network security world.

The two main categories of malware detection methods are static detection and dynamic detection. Static methods consist of analyzing the application without executing it and relying only on information that can be extracted from the apk file, such as the android manifest, API calls or disassembled low-level code. It is not only a safe way to analyze an application but it is also efficient, it has small overhead and the prediction methods it relies on are simple [6]. However, dynamic methods consist of running the application and observing its behaviour and thus, collecting more critical information, making it the preferred method. Dynamic malware detection also involves malware triggering. Infected applications run normally until the user inputs a series of events in the device (stimuli) and triggers the malicious piece of code [6]. In order not to infect the host system, dynamic analysis methods run the app in a safe environment. An app is labelled as malware by studying system parameters such as CPU or memory usage and feeding them to an existing classification algorithm. Although it may seem irresponsible to execute malware in a machine, the run-time data collected from it is what helps to determine the nature of an application. While there are a range of different dynamic methods available, two will be discussed within this study: MonkeyRunner and UI-guided.

MonkeyRunner utility provided by Google was originally intended for developers to stress test their app by sending random events. It is the most conventional tool to trigger malware as it is easy to use and requires no instrumentation. This acts as an advantage given that modern malware can detect if instrumentation is being used and prevents itself from triggering. Even though it is not proved, the random based input of events by MonkeyRunner and the unreasonable time it may take to trigger the malware could lead to the belief that it is an inferior tool against the UI-guided lightweight tool DroidBot. This tool also does not require instrumentation, as it generates a model of the app under testing and produces its next input, getting aid from information obtained during run-time. The reason why DroidBot is more lightweight is because it does not rely on any static analysis while it models the explored states. [7]

Proceeding from previous results obtained using MonkeyRunner [4], this thesis is a preliminary study to determine whether using a more sophisticated tool like DroidBot will produce notably different results.

In summary, a dataset of 8 apps was tested against both methods previously described. Data was collected during runtime and processed afterwards revealing that in fact in some features there is a difference but in others this difference is not as substantial.

This report is divided into three sections:

- **Background:** An explanation of the previous experiment and the environment that the new one inherits as well as a brief description of the dataset.

- **Approach:** The explanation of the scripts used for the execution of the new experiment and the difficulties that were encountered.

- **Results:** Evaluation of the results obtained and reaching the goal of the thesis - a comparison between the two experiments.

2 Background

This experiment is a continuation to the previous one regarded in the paper "Extinguishing Ransomware - A Hybrid Approach to Android Ransomware Detection" [4] written by advisor Alberto Ferrante. This paper addresses the use of a hybrid system with a static and a dynamic phase as a solution for malware detection. The static method essentially consists of using structural code analysis: Every application is pre-processed to obtain the opcode 2-grams sequences (grouping of 2 operational codes). Afterwards, the binary classifier gets trained with a labeled dataset and finally the classifier is used for determining whether or not the apps are malware. With this method, a precision ranging from 0.968 to 0.998 was obtained [4].

Similarly to the static method, the dynamic method also has two phases of pre-processing and learning. In this case however, the classifier was trained with execution records of apps with malicious behaviour, and then the results of the classification were used as the input for a sliding-window algorithm that classifies the applications considering past history of execution. Using different metrics, detection rate results ranged from 80.27% to 96.33% [4]. Combining both methods, the apps that were undetected by the static algorithm underwent dynamic analysis too. This hybrid method described in the paper obtained a 100% detection [4].

As the previous experiment used MonkeyRunner for the application exerciser, this is therefore an introductory experiment to check if using DroidBot for malware detection makes a considerable difference against MonkeyRunner.
Given the nature of this comparison, this project has inherited some constraints: every tool, resource or environment that was used for the previous experiment has to be used for this one as well, naturally with the exception of DroidBot.

In the following subsections the tool DroidBot, the environment in which the experiment took place and the dataset which was used will be discussed.

2.1 DroidBot

Developed in the Peking University, this tool fetches information for monitoring purposes as well as sending input events to the application; both tasks made via `adb`. These are lightweight processes because they rely on already built-in android utilities. The information that DroidBot fetches consist mainly on UI information, process information extracted from the `ps` command and the log the app outputs. Whereas for sending events, DroidBot replicates different kinds of stimuli such as UI inputs, intents or sensor data. As aforementioned, the tool produces a model of the app during run-time. This model essentially is a directed graph where states of the app are nodes and inputs that changed the state are arcs. The dynamic construction of this graph relies on a preexisting state comparison algorithm which considers two nodes as being not equal if their UI information is different [7]. The latest release of the tool was in 2017 but given the old version of the dataset and the AVDs, the previous release 1.0.0a2[1] was used for this project.

2.2 alarisrv10 Linux Server

The server on which the previous experiment was executed is a x86_64 Linux machine. The Android Software Development Kit available is the release 20140702(Android 4) and the emulator provided is the one inside this SDK. The reason for which such an old version is required is because the malware sample consists of apps with an Android 4 target [4]. However, due to incompatibility issues encountered during the experiment, DroidBot required a newer version of the SDK and thus, the emulator as well. Since the data extracted is inherent to the emulator, the decision of choosing a newer version of the emulator meant that the previous raw information files were different from the newer ones. This caused a significant setback because all the results from the MonkeyRunner experiment became unusable, meaning that the apps had to be tested again with the MonkeyRunner tool as well.

Another recurring tool included in the SDK that allows communication between the host and the emulator called `adb` was used to extract the needed data regarding CPU, memory and system calls. For the network data, all network traffic was dumped from the emulator which was then processed with the tool `tcpstat`.
All these tools define the environment in which this experiment took place.

2.3 Dataset

The specific type of malware that is going to be experimented on is called ransomware. Its main goal is to block the device or encrypt its data for a later proposition of a ransom to the user to get their data back. As an example, a few years back the FBI MoneyPak Ransomware[2] infected many devices. Posing as the FBI the app prompted messages containing serious allegations and accusations of breaking the law and then stated that a fine had to be paid. The original dataset of the previous experiment included 672 ransomware apps obtained from the HelDroid[3] dataset belonging to the time period between December 2014 and June 2015. For the purpose of this experiment a smaller subset of 8 apps from the sample was chosen.

3 Approach

To achieve success with this experiment, a sample of tested apps is needed in order to get the system features results. Therefore, a large part of the workload was to make the tool DroidBot work and run the execution scripts after adjusting them to be compatible with the new tool. As mentioned before, due to the change in the emulator version the already obtained results in the hybrid method experiment were useless. This meant that system features results had to be extracted using the two tools MonkeyRunner and DroidBot.

The main subject of this section is detailing how the scripts for execution and data extraction work and how the tool DroidBot was made to function properly in the environment provided.

3.1 behave_lugano.sh

While the tool DroidBot has to be executed over an app, the system data has to also be extracted during its run-time; the bash script `behave_lugano.sh` is the responsible for this task. Originally co-written by thesis advisor Alberto Ferrante this script was modified to make it work with the tool DroidBot. Therefore, the way the original version works and what other scripts are needed

to be executed needs to be explained first:

The main script is `behave_lugano.sh`, however the one that initiates the whole process of testing the apps is called `script.sh`. The purpose of this script is to simultaneously run 4 instances of the emulator, each of which are tested by a `behave_lugano.sh` process. In order to launch an emulator instance, the script `startemu.sh` is executed; it starts an android emulator process indicating the `avd`, the port for the `adb` connection and the path for the network dump file. Once the emulator is set, `script.sh` invokes a process of `behave_lugano.sh` passing by argument the name of the app and the emulator instance id. x What the main script does is:

- Waits for the device to be ready and installs the apk in it via `adb`.
- Extracts the package name which will be necessary for future tasks.
- Initiates the process of MonkeyRunner in the device indicating the package name of the app to be tested and the total count of stimuli (In this case 20,000)
- Every two seconds until MonkeyRunner finishes the script does the following:
  - Saves the time stamp of the current time.
  - Extracts data to `.txt` files about CPU Usage and memory executing in the `adb` shell the command `dumpsys`.
  - Extracts data to `.txt` files about the execution trace executing in the `adb` shell the command `strace`.
  - Checks if the process id of the app changed and updates the local variable storing the pid if necessary. This is necessary because the `strace` command needs the pid as an argument.
- Terminates the emulator

After the `behave_lugano` process finishes, the `script.sh` process rewrites the `avd` files so emulators are always launched from fresh copies. Then it initiates the cycle again with the next apk to be tested.

In order to make the script `behave_lugano.sh` work with DroidBot some changes had to be made. The first observation was that DroidBot installs the application before it starts sending the events, therefore the step of installing the app via `adb` needed to be removed. This provoked another issue in which the pid of the app in the device was unknown until DroidBot finished to install and launch the application, resulting in empty records at the beginning of each raw data text file. Further investigations should be made to determine if this change influences the results. The stop condition had to change as well to 15 minutes spent executing for the reason that DroidBot is not able to input 20,000 stimuli in an acceptable amount of time because between events, the tool runs an algorithm to determine which is the best input to send next.

One other last small addition to the script was made; in some cases, during the execution of DroidBot a message saying "This device might have been hijacked" was prompted. In these cases a file called "`hijacked.txt`" was created as an indicator that said the message was prompted. Even though for the purposes of this experiment this message was discarded, it may be interesting to consider in future works.

3.2 summary.sh

After the raw data text files are extracted they need to be processed in order for the results to be compared efficiently. Firstly, a script called `clean.sh` needs to be invoked to remove any incomplete results and to change the formatting using the tool `dos2unix`, then the actual parsing can be done. The main script `summary.sh` is responsible for creating a `.csv` file for each app tested. The four different categories of system features have their own script for parsing the correspondent `.txt` file so in total these are the following scripts:

- `summary-cpu.sh`
- `summary-mem.sh`
- `statistics-calls.sh`
- `statistics-network.sh`

The four scripts function similarly in the way that all of them produce a single `.csv` file using a pearl script to parse the `.txt` that looks for the features that need to be extracted. The only difference between the scripts is that the ones for calls and network have to adjust the records produced by `tcpstat` and `strace` respectively to the time stamps saved by `behave_lugano.sh`.

However, due to the fact that a more recent version of the emulator had to be used, a newer version of the scripts had to be used as well. The structure and functionality of the scripts are the same as the described above, however, some modifications had to be made like changing the name of the features extracted.

3.3 Sequence of events

The first step of this study was to install and setup the tool DroidBot in the server. The tool had to be installed with `pip`, but the existing version in the server was incompatible. Due to the lack of privileges in the server, a newer version of python itself and some python related tools like `pip` or `setuptools` were installed with a tool called `spack`. Briefly, this utility is a package manager that installs packages with every dependency, meaning that the lack of privileges presented no further problems since the designated user had permission over every other package.

Having two versions of python caused some troubles which were solved using another python tool called `virtualenv`. It creates a virtual environment with a fresh copy of python and some of its packages. It turned out to be a rather useful tool during the installation phase because different versions of DroidBot were tested and `virtualenv` allowed them to coexist without any issues.

After the environment is setup and tested properly, the apps need to be exercised. The two versions of the script `behave_lugano.sh` described in the previous section were executed for every app in the dataset producing two different sets of results, one for Monkey and the other for DroidBot.
The raw data obtained was processed with the extraction scripts described in the previous section and the only work left was to compare the results.

4 Evaluation

The features extracted from the raw data logs contain information about the CPU usage, Memory, System calls and Network data. A smaller subset from the previous experiment was chosen for the reason aforementioned that different scripts were used for the data extraction. The following table details every feature that was extracted [4].

Category		Feature names
CPU	CPU Usage	Total CPU Usage, User CPU Usage, Kernel CPU Usage
	Virtual Memory	Page minor faults
Memory	Native Memory	Native Pss, Native Private Dirty, Native Private Clean, Native Swapped Dirty, Native Heap Size, Native Heap Alloc,Native Heap Free
	Dalvik Memory	Dalvik Pss, Dalvik Private Dirty,Dalvik Private Clean, Dalvik Swapped Dirty, Dalvik Heap Size, Dalvik Heap Alloc, Dalvik Heap Free, Dalvik Other Pss, Dalvik Other Private Dirty, Dalvik Other Private Clean, Dalvik Other Swapped Dirty
	Stack	Stack Pss, Stack Private Dirty, Stack Private Clean, Stack Swapped Dirty
	Android Shared Memory	Ashem Pss, Ashem Private Dirty, Ashem Private Clean, Ashem Swapped Dirty
	Other memory and mapped files	Other dev Pss, Other dev Private Dirty, Other dev Private Clean, Other dev Swapped Dirty, .so mmap Pss, .so mmap Private Dirty, .so mmap Private Clean, .so mmap Swapped Dirty, .apk mmap Pss, .apk mmap Private Dirty, .apk mmap Private Clean, .apk mmap Swapped Dirty,.dexmmap Pss, .dexmmap Private Dirty, .dexmmap Private Clean, .dexmmap Swapped Dirty, Other mmap Pss, Other mmap Private Dirty, Other mmap Private Clean Other mmap Swapped Dirty
	Memmory mapped fonts	.ttfmmap Pss, .ttfmmap Private Dirty, .ttfmmap Private Clean, .ttfmmap Swapped Dirty
	Non-classified memory allocations	Unknown Pss, Unknown Private Dirty, Unknown Private Clean, Unknown Swapped Dirty
	Memory Totals	TOTAL Pss, TOTAL Private Dirty, TOTAL Private Clean, TOTAL Swapped Dirty, TOTAL Heap Size, TOTAL Heap Alloc, TOTAL Heap Free
Statistics on System calls		Number of Syscalls, No. of different syscalls, Average no. of calls per syscall, No. of calls occuring once, No. of calls occurring multiple times
Network	Link layer networking	Number of ARP packets, AVG. PKT Size bytes, bps, Number of ICMP packets, Size in byte standard deviation
	Internet layer networking	Number of IPv4 packets, Network load over last minute, Maximum packet size in bytes, Minimum packet size in bytes, Number of bytes, Number of packets, Number of packets per second, Number of IPv6 packets
	Transport layer networking	Number of TCP packets, Number of UDP packets

4.1 Comparison

This experiment consists of two time series with a 2 second observation period, one for MonkeyRunner and the other one for DroidBot. The average and standard deviation is computed for every feature, and then both the DroidBot average and standard deviation is compared with the Monkey average and standard deviation using the difference percentage. The formula used is:

$$\%difference = \frac{(Monkey_{val} - DroidBot_{val})}{DroidBot_{val}} \cdot 100$$

If the result is positive it means that the value of Monkey is higher than DroidBot, if it is negative it means the contrary. This formula was chosen because the percentage is more representative of the data but it is worth noting that it presents a fault when one of the values is 0. In the case where the DroidBot average equals 0 the precentage would go to ∞.

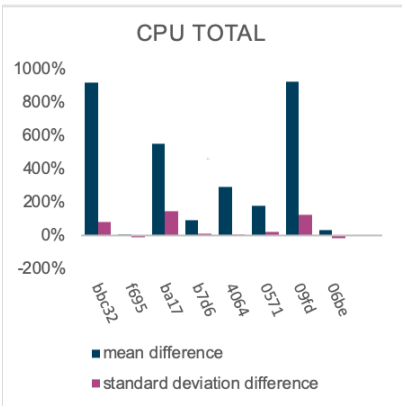
4.2 Results

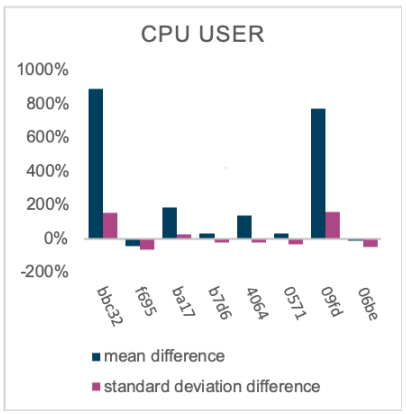
The following section includes the results obtained after comparing the execution traces from both experiments. Not every feature is displayed or discussed as in some of the features no significant change could be noted. However, for the rest of them, the data is analyzed with the help of plots and a possible explanation for their results is given.

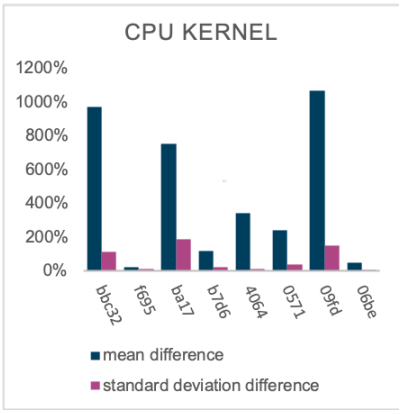
On the title of each graph appears the name of the feature plotted. The X axis in the plots represent the apps tested and in the labels the first characters of the hash of the real name of the app is displayed. The Y axis represents in percentage, the difference in percentage in both mean and standard deviation. There is a possibility for negative values to appear for the reason mentioned before: a negative value means that the DroidBot value for that feature is larger than the MonkeyRunner.

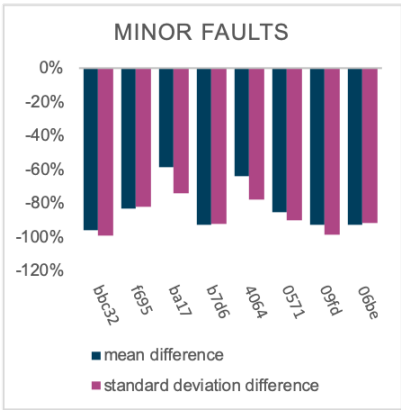
4.2.1 CPU

As it is highlighted in the graphs, the differences in the CPU are notable. The explanation for this result could be the difference in how both of the tools work. As mentioned before, MonkeyRunner randomly sends events with a very high frequency, while in DroidBot the time between events is higher, which results in an increased CPU usage for the former tool.

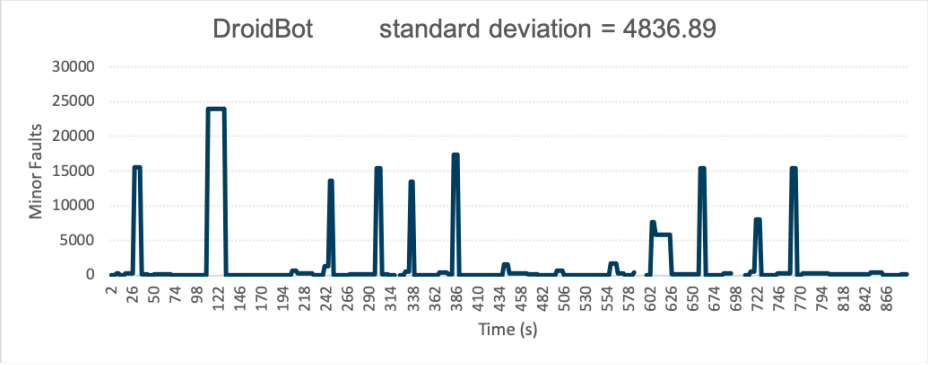


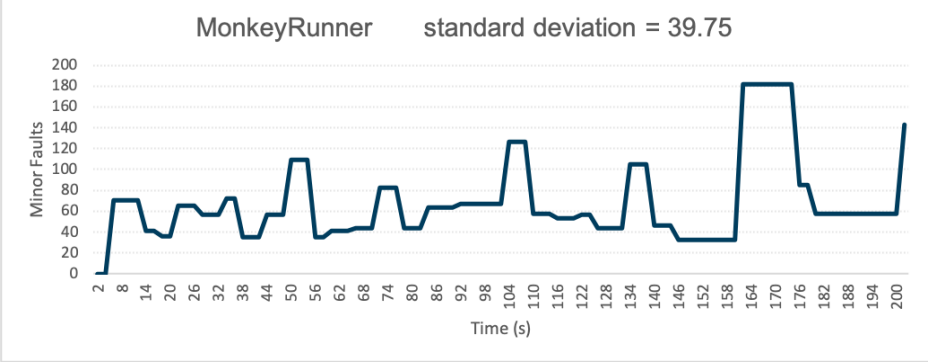






For every feature it can be observed that Monkey has a significantly higher CPU usage with values reaching a 1000% growth. However, for the minor faults feature it appears to be the opposite. A possible explanation for this result could be that while using DroidBot, the more time between events allows for other processes to overwrite the cache and when the app that is undergoing the test executes the next instruction it produces a cache miss. In order to understand better this last feature, for both tools, a graph representing the minor faults over time for the app "bbc32" are depicted.



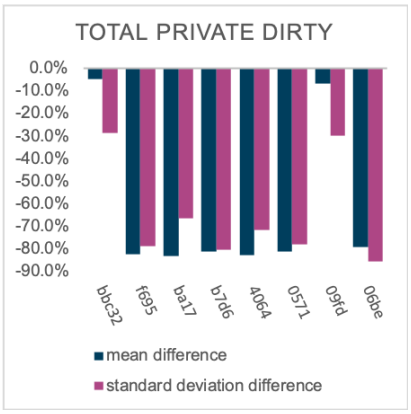


Now, the significant difference in the standard deviation can be appreciated. Multiple spikes in minor faults appear in the DroidBot graph, this could happen due to the reason mentioned multiple times that the time between events for the tool DroidBot is higher so the application under test have some idle periods.

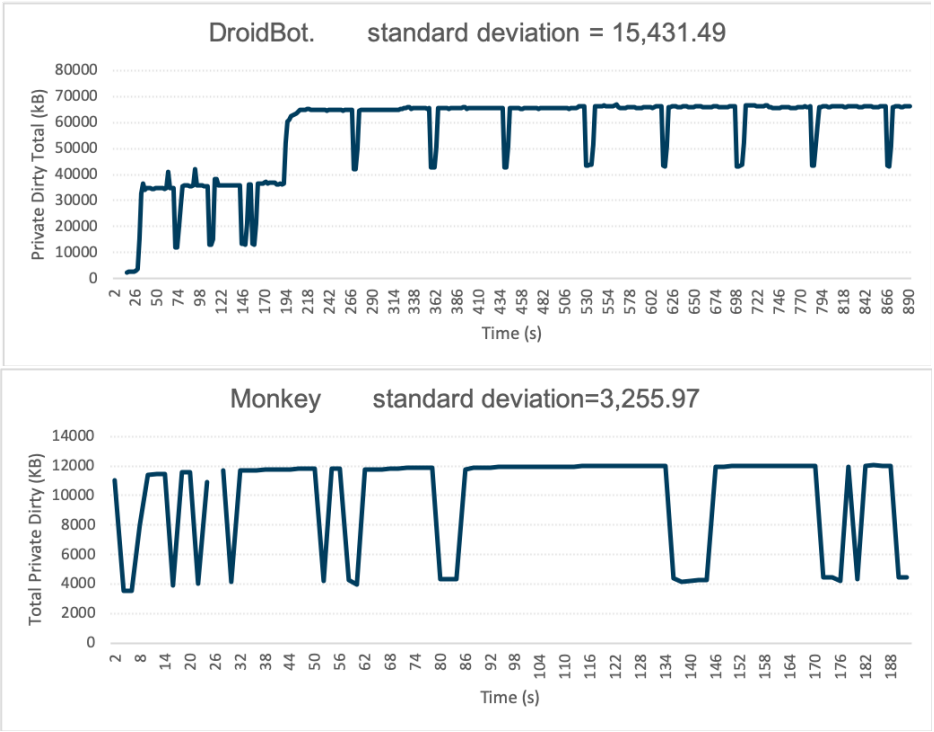
4.2.2 **Memory**

In the vast majority of the memory features the values were negative and did not surpass the -10% mark. This means that consistently DroidBot produces higher values for memory features. The largest values were observed in the "Total Private Dirty" and "Total Private Clean" features

reaching the -100% mark. Only the "Total Private Dirty" plot is featured because the latter one had a value high enough to make the rest of the plot unreadable.



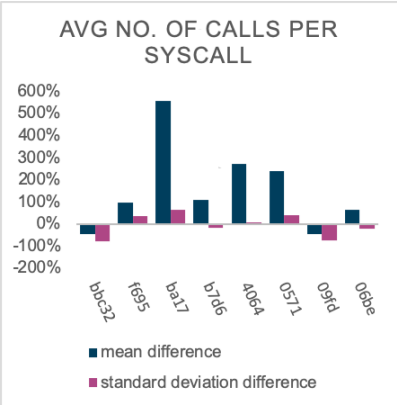
This graph is further explored for the app named "f695", same as before, for both tools, the feature is plotted over time.

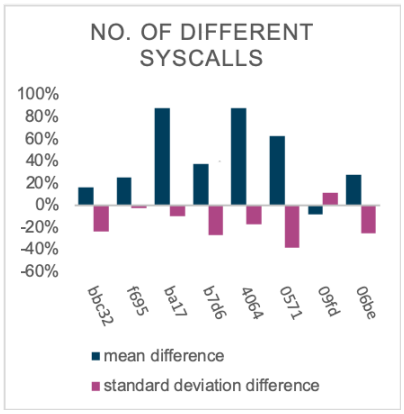


4.2.3 System Calls

Most of the features involving system calls represent a total number of system calls during the execution. Since the execution time of an app under DroidBot is considerably lower than the time under MonkeyRunner, the total number of system calls is inevitably going to be lower as well. For this reason most of the features are not plotted since the results are predictable. Graphs of two of the features that were most different are shown.

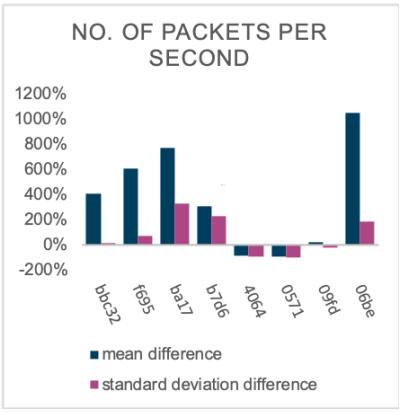
It can be highlighted that in the left plot one app shows a spike in the mean difference. As it is the only app with this behaviour it will be considered not representative.

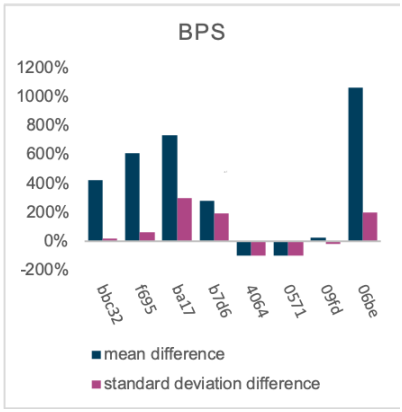




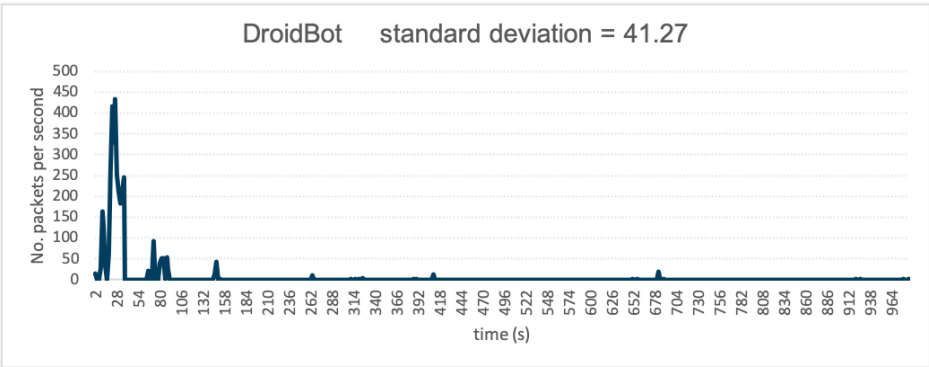
4.2.4 Network

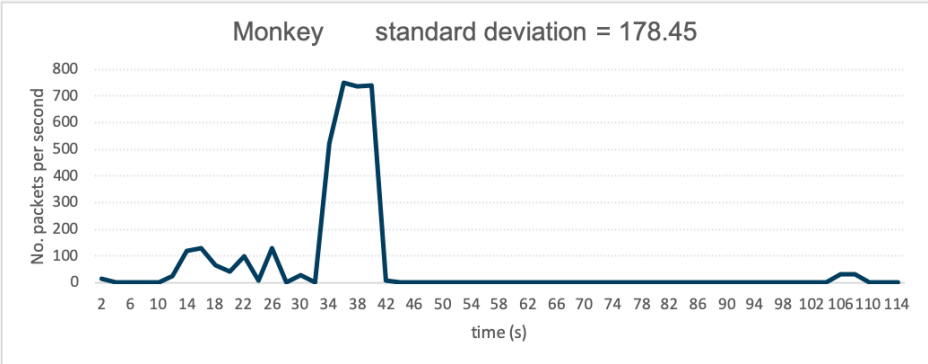
Network results in a way are very similar to the CPU ones. The higher activity during run-time with MonkeyRunner makes a notable difference in network features such as the number of packets sent or the number of bytes sent. This only corroborates that DroidBot is significantly less resource oriented. Two features are displayed:

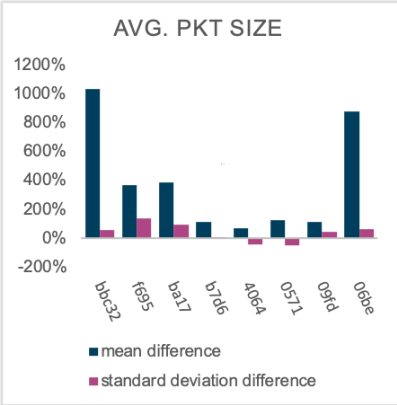




The results from the plots are expected. Even though for two apps DroidBot produced more network traffic, it can be stated that MonkeyRunner is the one with the higher value in network statistics. The difference in the standard deviation is complemented with graphs of the number of packets per second for the app named "ba17" for both tools.







The feature of the average packet size displayed in the graph shows an interesting result: it would be normal to think that the average packet size should be similar from both tools but this data depicts the opposite result and with considerably high numbers. A possible explanation could be that at the same time MonkeyRunner produces a higher traffic it simultaneously keeps the connections open for longer. Which could mean that the smaller protocol header packets have more presence in the average, however this is just a hypothesis and further investigation should be pursued.

5 Conclusion

Malware triggering/detection is an important aspect of application safety as it improves security. Even though dynamic methods of malware detection can be tedious and time consuming they also can make a difference while testing apps. It was previously tested that the MonkeyRunner method was efficient for ransomware detection and with this experiment the question of "Will DroidBot produce different results?" was raised.

A dataset of malicious applications was tested under both tools MonkeyRunner and DroidBot. The raw data collected by the bash scripts while the tools were working was processed into interpretable .csv files. With these files, the features that showed more change were plotted and interpreted. However, working with DroidBot was at times unreliable. This could be due to the combination of using an older version of DroidBot and the applications under test. The tool is also more sophisticated than Monkeyrunner which could have effected its reliability.

The results obtained ensured that there is a difference in some parameters such as CPU or Network while in others it was more limited. A statement which determined that DroidBot is a better tool than Monkey for malware detection cannot be made; a more in depth analysis using classifiers should take place in order to find out which one produces better results.

Moreover, this study can act as a primary reference for future experiments to be conducted in

determining whether DroidBot is a more useful tool for malware detection.

References

[1] https://github.com/honeynet/droidbot/releases.

[2] https://en.wikipedia.org/wiki/FBI_MoneyPak_Ransomware.

[3] http://ransom.mobi.

[4] Ferrante a., malek m., martinelli f., mercaldo f., milosevic j. (2018) extinguishing ransomware - a hybrid approach to android ransomware detection. in: Imine a., fernandez j., marion jy., logrippo l., garcia-alfaro j. (eds) foundations and practice of security. fps 2017. lecture notes in computer science, vol 10723. springer, cham.

[5] Z. Whittaker. New android adware found in 200 apps on google play. https://techcrunch.com/2019/03/13/new-android-adware-google-play/.

[6] R. C. Ying-Chih Shen and S.-H. Hung. Toward efficient dynamic analysis and testingfor android malware.

[7] Y. G. X. C. Yuanchun Li, Ziyue Yang. Droidbot: A lightweight ui-guided test inputgenerator for android.